Speculative Precomputation: Long-range Prefetching of Delinquent Loads

Jamison D. Collins[†], Hong Wang[‡], Dean M. Tullsen[†],
Christopher Hughes[§], Yong-Fong Lee[II], Dan Lavery[II], John P. Shen[‡]

[†]Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114

[‡]Microprocessor Research Lab
[II]Microcomputer Software Lab
Intel Corporation
Santa Clara, CA 95052-8119

[§]Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Abstract

This paper explores Speculative Precomputation, a technique that uses idle thread contexts in a multithreaded architecture to improve performance of single-threaded applications. It attacks program stalls from data cache misses by pre-computing future memory accesses in available thread contexts, and prefetching these data. This technique is evaluated by simulating the performance of a research processor based on the Itanium^{TM} ISA supporting Simultaneous Multithreading. Two primary forms of Speculative Precomputation are evaluated. If only the non-speculative thread spawns speculative threads, performance gains of up to 30% are achieved when assuming ideal hardware. However, this speedup drops considerably with more realistic hardware assumptions. Permitting speculative threads to directly spawn additional speculative threads reduces the overhead associated with spawning threads and enables significantly more aggressive speculation, overcoming this limitation. Even with realistic costs for spawning threads, speedups as high as 169% are achieved, with an average speedup of 76%.

1. Introduction

Memory latency still dominates the performance of many applications on modern processors, despite continued advances in caches and prefetching techniques. This problem will only worsen as CPU clock speeds continue to advance more rapidly than memory access times, and as the data working sets and complexity of typical applications increase. One approach to overcome this has been to attempt to overlap stalls in one program with the execution of useful instructions from other programs, using techniques such as Simultaneous Multithreading (SMT) [19, 20] as implemented in the Alpha 21464 [5]. The SMT techniques can improve overall instruction throughput under a multiprogramming workload; however, it does not directly improve performance when only a single thread is executing.

We propose Speculative Precomputation (SP) as a technique to improve single-thread performance on a multithreaded architecture. It utilizes otherwise idle hardware thread contexts to execute speculative threads on behalf of the non-speculative thread. These speculative threads attempt to trigger future cache miss events far enough in advance of access by the non-speculative thread that the memory miss latency is avoided entirely. Speculative precomputation could be thought of as a special prefetch mechanism that effectively targets load instructions that traditionally have been difficult to handle via prefetching, such as loads that do not exhibit predictable access patterns and chains of dependent loads.

To limit the increase in contention for fetch, execute, and memory system bandwidth from these speculative threads, SP is targeted only at the static loads that cause the most stalls in the non-speculative thread, which we call delinquent loads. We find that in most programs the set of delinquent loads is quite small; commonly 10 or fewer static loads cause more than 80% of L1 data cache misses. Similar observation has been made in [1].

Speculative threads execute precomputation slices (p-slices), which are sequences of dependent instructions which have been extracted from the non-speculative thread and compute the address accessed by delinquent loads. When a speculative thread is spawned, it precomputes the address expected to be accessed by a future delinquent load, and prefetches the data. Speculative threads can be spawned

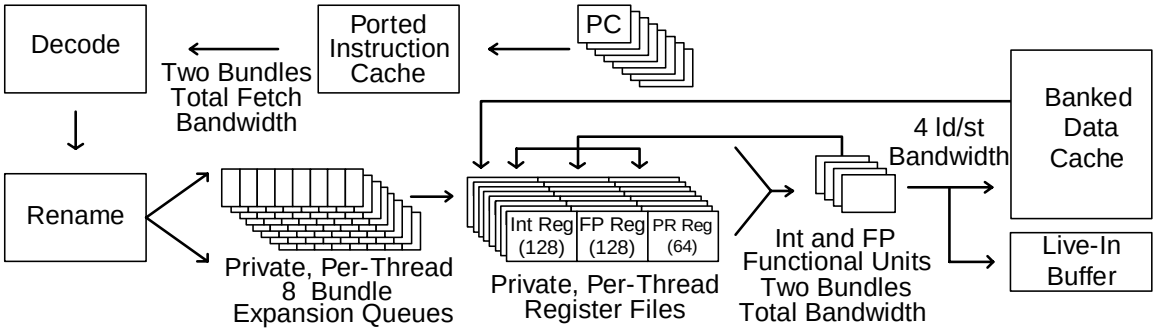

Figure 1. Pipeline organization of a research Itanium processor with SMT support.

when a designated instruction from the non-speculative thread reaches a particular stage in the pipeline (e.g, the commit stage) or by one speculative thread explicitly spawning another. These conditions are referred to as a basic trigger and a chaining trigger, respectively. Though not explored here, further advanced trigger types could be used, such as those described in [23].

This work evaluates the performance gains provided by both forms of speculative precomputation on a research SMT processor implementing the Itanium^{TM} instruction set [8]. Our results show that under ideal conditions, SP with basic triggers provides gains as high as 30%, but under realistic hardware assumptions these gains are reduced significantly. However, using chaining triggers in addition to basic triggers on a processor with 8 total thread contexts achieves a speedup of up to 169% (average 76%), even with realistic hardware assumptions.

The remainder of the paper is organized as follows. Section 2 discusses related research. Section 3 presents our baseline processor model and outlines the evaluation methodology. Section 4 provides motivation for SP. Section 5 explains algorithms for extracting and optimizing p-slices and mechanisms employed by SP at runtime. Section 6 provides experimental results for SP when only basic triggers are used. Section 7 shows results when both basic triggers and chaining triggers are used. Section 8 concludes.

2. Related Work

Various research projects have considered leveraging idle multithreading hardware to improve single-thread performance. Roth and Sohi proposed speculative data driven multithreading (DDMT) [13], in which speculative threads execute on idle hardware thread contexts to prefetch for future memory accesses and predict future branches. Their work focused on application to an out-of-order processor. In this paper, we first evaluate an idea motivated by Roth's research in the basic trigger scheme to gauge its potential for the Itanium architecture and to identify areas for further improvement. This work focuses on a research Itanium family SMT processor in which the core pipeline is in-order.

Zilles and Sohi studied the backward slices of performance degrading instructions [25]. Their work focused on characterizing the instructions preceding hard-to-predict branches or cache misses and on exploring techniques to minimize the size of the backward slices. The precomputation slices used by our work are constructed within an instruction window of size 128-256, in a similar manner to Zilles and Sohi's work, which assumed a window size of 512.

Chappel et al. proposed Simultaneous Subordinate Microthreading (SSMT) [4], in which sequences of microcode are injected into the main thread when certain events occur, providing a software mechanism to override default hardware behavior, such as branch predictor algorithms. Dubois and Song proposed Assisted Execution [15] in which tightly-coupled subordinate threads, known as nanothreads, share fetch and execution resources on a dynamically scheduled processor to accelerate or gather performance statistics on the main thread.

Sundaramoorthy et al. proposed Slipstream Processors [17], in which a non-speculative version of a program runs alongside a shortened, speculative version. Outcomes of certain instructions in the speculative version are passed to the non-speculative version, providing a speedup if the speculative outcome is correct. Their work focused on implementation on a chip-multiprocessor (CMP).

Wallace et al. proposed Threaded Multipath Execution (TME) [22]. TME attempts to reduce performance loss due to branch mispredictions by forking speculative threads that execute both directions of a branch, when a hard to predict branch is encountered. Once the branch direction is known, the incorrect thread is killed.

This work is unique in its targeting of the Itanium family processor, the low hardware cost of our thread spawning mechanisms, and the use of chaining triggers to greatly increase the effectiveness of these techniques.

3. Experimental Methodology

This paper studies the effects of Speculative Precomputation on a research SMT processor implementing the Itanium [7] instruction set architecture. The pipeline organization is depicted in Figure 1. Processors in the Itanium family fetch instructions in units of bundles, rather than individual

Pipeline Structure	2GHz: 8 stage pipeline, 1 cycle misfetch penalty, 6 cycle mispredict penalty 4GHz: 10 stage pipeline, 1 cycle misfetch penalty, 8 cycle mispredict penalty 8GHz: 12 stage pipeline, 2 cycle misfetch penalty, 10 cycle mispredict penalty
Fetch	2 bundles from 1 thread, or 1 bundle from 2 threads
Branch Predictor	2K entry GSHARE, 256 entry 4-way associative BTB
Expansion Queue	Private, per-thread, in-order 8 bundle queue
Register Files	Private, per-thread register files. 128 Int Reg, 128 FP Reg, 64 Predicate Reg
Execute Bandwidth	Up to 6 instructions from one thread or up to 3 instructions from 2 threads
Memory Hierarchy	L1 (separate I and D): 16K 4-way, 8 way banked, 1 cycle latency L2 (shared): 256K 4-way, 8 way banked, 7 cycle latency L3 (shared): 3072K 12-way, 1 way banked, 15 cycle latency All caches have 64 byte lines
Memory latency	2GHz: 115 cycles, 4GHz: 200 cycles, 8GHz: 357 cycles
TLB Miss Penalty	2GHz: 30 cycles, 4GHz: 60 cycles, 8GHz: 120 cycles

Table 1. Details of the modeled research Itanium processor

instructions [8]. Each bundle is comprised of three independent instructions that the compiler has grouped together. The modeled processor has a maximum fetch bandwidth of two bundles per cycle.

Instructions are issued in-order, from an 8-bundle expansion queue, which operates like an in-order instruction queue. The maximum execution bandwidth is 6 instructions per cycle, which can be from up to two bundles. Sufficient functional units exist to guarantee that any two issued bundles are executed in parallel without functional unit contention, and up to four loads or stores can be performed per cycle.

The aggressive memory hierarchy consists of separate 16K 4-way set associative L1 Instruction and Data caches, a 256K 4-way set associative L2 shared cache and a 3072K 12-way set associative shared L3 cache. All caches are on chip. Data caches are multi-way banked, but the instruction cache is dual ported to avoid fetch conflicts between threads. Note: even on a processor with more than 2 thread contexts, dual- ported I-Cache is assumed. More details are described in the next subsection. Caches are non-blocking with up to 16 misses in flight at once, where multiple misses to the same cache line each count separately. A miss upon reaching this limit stalls the execute stage. Speculative threads are permitted to issue loads that will stall the execute stage.

We model a pipelined hardware TLB miss handler [14]. It resolves TLB misses by fetching the TLB entry from an on-chip buffer (separate from data and instruction caches). In the default configuration, TLB misses are handled in 30 clock cycles, and we allow memory accesses from speculative threads to initiate TLB update.

The baseline processor has a 2GHz clock rate. Higher clock frequencies are modeled by increased latency to main memory, extra overhead for TLB miss handler and longer penalties for both branch misprediction and instruction misfetch. On-chip cache latencies remain constant in terms of CPU cycles. Unless otherwise noted, simulations assume a 2GHz processor configuration. Full details of the modeled processor are shown in Table 1.

Suite	Benchmark	Input	Fast-forward
SPECFP	art	Training	1 billion
SPECFP	equake	Training	1 billion
SPECINT	gzip	Training	1 billion
SPECINT	mcf	Training	1 billion
Olden	health	5 Levels	100 million
Olden	mst	1031 nodes	230 million

Table 2. Workload Setup

3.1. Multithreading

All simulations in this work assume a single non-speculative thread persistently occupies one hardware thread context throughout its execution while the remaining hardware thread contexts are either idle or occupied by speculative threads. The term “non-speculative thread” will be used interchangeably with “main thread” throughout this paper. Each hardware thread context has a private, per-thread expansion queue and register files. All architecturally visible registers, including 128 general purpose integer registers (GR), 128 fp registers (FR), 64 predicate registers (PR) and 128 control registers [14] are replicated for each thread.

If more than one thread is ready to fetch or execute, two threads are selected from those that are ready, and each is given half of the resource bandwidth. Thus, if two threads are ready to fetch, each is allowed to fetch one bundle. A round-robin policy is used to prioritize the sharing between threads. If only one thread is ready, it is allocated the entire bandwidth.

If instructions stall before they reach the expansion queue, the stall will cause pipeline backpressure. To prevent a stalling thread from affecting all other threads, a fetch-replay is performed when a thread attempts to insert a bundle into its already full queue. When this occurs, the bundle is dropped

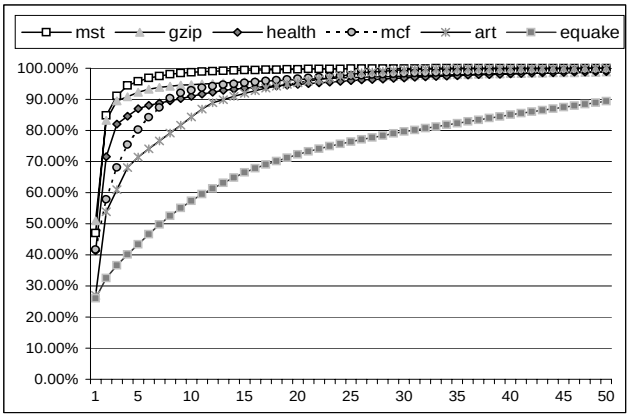

Figure 2. Cumulative L1 data cache misses contributed by the worst behaving static loads.

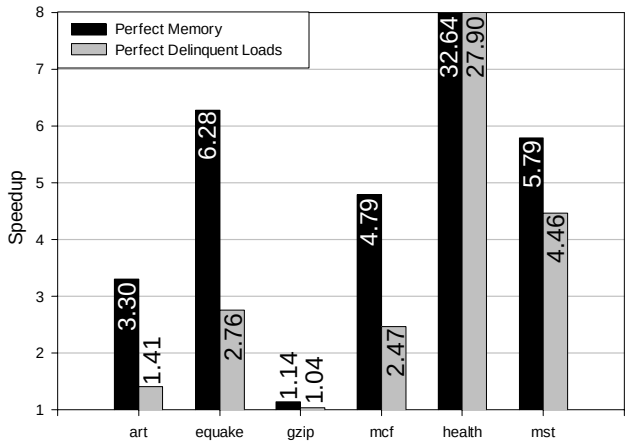

Figure 3. Speedup when 10 worst behaving static loads are assumed to always hit in cache.

by the expansion queue, and the thread of concern is prevented from fetching again until it has issued an instruction.

3.2. Simulation Environment and Workloads

We model processor performance using a version of the SMTSIM simulator [18] that has been enhanced to work with Itanium binaries. SMTSIM is a cycle-accurate, execution-driven simulator of SMT processors. Benchmarks for this study include both integer and floating point benchmarks selected from the CPU2000 suite [16] and pointer-intensive benchmarks from the Olden suite [3]. Benchmarks are selected because their performance is limited by poor cache performance or because they experience high data cache miss rates. The benchmarks and simulation setup are summarized in Table 2. Unless otherwise noted, all benchmarks are simulated for 100 million retired instructions after fast-forwarding past initialization code (with cache warmup). In our initial simulation experiments, much longer runs of the benchmarks were performed, however it was observed that the longer-running simulation results yielded only negligible performance differences.

All binaries used in this work are compiled with the Intel Electron compiler for the Itanium architecture [2, 11]. This advanced compiler incorporates the state-of-the-art optimization techniques known in the compiler community as well as novel techniques designed specifically for the features of the Itanium architecture. Benchmarks for this research are compiled with maximum compiler optimizations enabled, including those based on profile driven feedback, such as aggressive software prefetching, software pipelining, control speculation and data speculation.

4. Delinquent Loads

For most programs, only a small number of static loads are responsible for the vast majority of cache misses [1]. Figure 2 shows the cumulative contributions to L1 data cache misses by the top 50 static loads for the processor modeled in this research, running benchmarks to completion. It is evident that cache misses in these programs are dominated by a few static loads. We call these poorly behaved loads delinquent loads.

In order to gauge the impact of these loads on performance, Figure 3 compares results when the worst 10 delinquent loads are assumed to always hit in the L1 cache, versus a perfect memory subsystem where all loads hit in the L1. In most cases, eliminating performance losses from only the delinquent loads yields much of the speedup achievable by zero-miss-penalty memory. This data motivates special focus on a technique that targets these delinquent loads.

5. Speculative Precomputation

In speculative precomputation, an event triggers the invocation and execution of a p-slice. A p-slice is a speculative thread that computes and prefetches an address expected to be accessed by a delinquent load in the near future. Speculative threads are spawned under one of two conditions: when encountering a basic trigger, which occurs when a designated instruction in the main thread reaches a particular pipeline stage (such as the commit stage), or a chaining trigger, when one speculative thread explicitly spawns another.

A speculative thread is spawned by allocating a hardware thread context, copying necessary live-in values into its register file, and providing the thread context with the address of the first instruction of the thread. If a free hardware context is not available the spawn request is ignored.

Necessary live-in values are always copied into the thread context when a speculative thread is spawned. This eliminates the possibility of inter-thread hazards, where some register is overwritten in one thread before a child thread has read it. Fortunately, as shown in Table 3, the number of live-in values that must be copied is very small.

When spawned, a speculative thread occupies a hardware thread context until the speculative thread completes execu-

Benchmark	# Slices(Opt)	Average Length(Opt)	Average # Live-in
art	26 (2)	14.7 (4)	3.5
equake	35 (8)	13.1 (12.5)	4.5
gzip	307 (9)	14.0 (9.5)	6.0
mcf	49 (6)	5.2 (5.8)	2.5
health	33 (8)	15.9 (9.1)	5.3
mst	138 (8)	34.8 (26)	4.7

Table 3. Statistics on p-slices for delinquent loads. Numbers shown in parenthesis are values after optimization.

```
arc=arcs+group_pos;
for(;arc<stop_arcs;arc+=nr_group){
   if(arc->ident>BASIC){                                    Delinquent Load#1
      red_cost=arc->cost-arc->tail->potential+              Delinquent Load#2
                        arc->head->potential;               Delinquent Load#3
      if((red_cost<0&&arc->ident==AT_LOWER)||
         (red_cost>0&&arc->ident==AT_UPPER)){
         basket_size++;
         perm[basket_size]->a=arc;
         perm[basket_size]->cost=red_cost;
         perm[basket_size]->abs_cost=ABS(red_cost);
      }
   }
}
```

	L1 Miss Rate / % Capacity Miss	L2 Miss Rate / % Capacity Miss	L3 Miss Rate / % Capacity Miss
Delinquent Load# 1	99.95% / 99.98%	48.06% / 82.78%	67.64% / 97.38%
Delinquent Load# 2	80.92% / 97.60%	63.55% / 86.51%	20.04% / 47.88%
Delinquent Load# 3	93.10% / 99.1%	45.33% / 74.65%	20.70% / 44.74%

Figure 4. Sample procedure from MCF, pbeampp.c lines 180-195, containing 3 delinquent loads.

tion of all instructions in the p-slice. Speculative threads must not update the architectural state, for example, by executing a store instruction.

5.1. Speculative Precomputation Tasks

Several steps are necessary to employ speculative precomputation: identification of the set of delinquent loads, construction of p-slices for these loads, and the establishment of triggers. This work assumes that these steps are performed with some compiler assistance as well as some hardware support. Our future work will explore the actual implementation details of both compiler assistance and hardware support needed. The following section gives details on each phase of the procedure, followed by a demonstration of the procedure when applied to a delinquent load from the CPU2000 benchmark mcf, in Section 5.2.

Optimize Basic Triggers and P-Slices Many of the identified p-slices can be removed. These include redundant triggers (multiple triggers targeting the same load), rarely-executed triggers, and triggers that are too close to the target load. Table 3 shows that most potential p-slices are actually removed in this phase. Additionally, generated slices are modified to make use of induction unrolling [13].

Identify Delinquent Loads The set of delinquent loads that contribute the majority of cache misses is determined through memory access profiling, performed either by the compiler or a memory access simulator, such as dinero [10]. From this profile analysis, the loads that have the largest impact on performance are selected as delinquent loads. This work uses the total number of L1 cache misses as the criterion to select delinquent loads, but other filters (e.g., one that also accounted for L2 or L3 misses or total memory latency) could also be used.

Construct P-Slices In this phase, each benchmark is simulated on a functional Itanium simulator [21] to create the p- slices for each delinquent load. Whenever a delinquent load is executed, the instruction that had been executed 128 instructions prior in the dynamic execution stream is marked as a potential basic trigger. The next few times that this potential trigger is executed, the instruction stream is observed to verify the same delinquent load will be executed somewhere within the next 256 instructions. If the potential trigger consistently fails to lead to the delinquent load, it is discarded. Otherwise, if the trigger does consistently lead to the delinquent load, the trigger is confirmed and the backward slice of instructions between the delinquent load and the trigger is captured. This work considers a smaller window of instructions from which to generate the p-slice than previous work [25, 13] in anticipation of efficient hardware implementation. Instructions (limited to maximum of 256) between the trigger and the delinquent load constitute potential instructions for constructing the p-slice. Those unnecessary to compute the address accessed by the delinquent load are eliminated, resulting in small p-slices generally between 5 to 15 instructions in length.

Link Slices into Binary For each benchmark, the instructions from each p-slice are appended to the program binary in a special program text segment. Steps can be taken to minimize potential instruction cache interference between the speculative thread and the main thread [24]. However, for this study we found instruction fetch for the p-slices did not introduce any noticeable increase in I-cache misses for the non-speculative thread.

5.2. An Example – MCF

Figure 4 illustrates the source code for a key loop from the mcf benchmark. The loop contains three delinquent loads, which are annotated, and their cache miss statistics shown. It is important to note the high number of cache misses at both L1 and L2 are due to capacity misses [6]. Conventional wisdom dictates that we increase the size of the cache, however, this risks impacting the cache access latency.

Figure 5 shows a partial assembly listing and a p-slice captured from this procedure when applied to delinquent load #3 in Figure 4. The p-slice targets an instance of

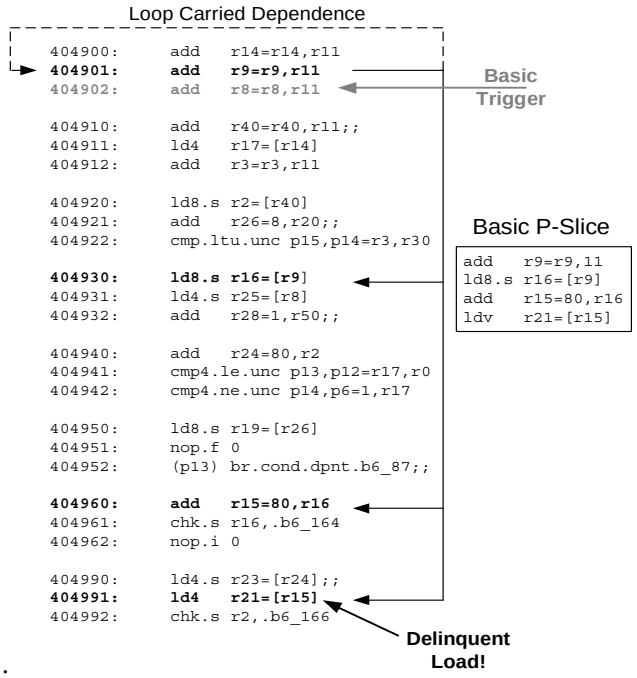

Figure 5. The p-slice for an example delinquent load.

the delinquent load one loop iteration ahead of the non-speculative thread when the p-slice is spawned.

It is important to note that simply embedding the instructions from the p-slice directly in the main program as a form of software prefetch will not be as effective as SP, and in some circumstances could actually lower performance. If the first load misses in cache and the add which follows attempts to access this data before it has arrived, the main thread will stall. In contrast, when executed as a speculative thread, the main thread is unaffected by any stalls that occur when executing a p-slice.

5.3. Comparison to Traditional Prefetching

We briefly highlight the key differences between speculative precomputation and traditional forms of prefetching. Software prefetching places explicit prefetch instructions in the main program code in advance of loads. These techniques are less effective in the presence of irregular control flow or load misses that depend on other load misses (particularly in an in-order processor, where a load miss can stall the processor, even if it is only computing an address to be used in a later prefetch). Hardware prefetching techniques work best on regular data accesses; hardware techniques that do well with pointer-chasing codes have been proposed [12, 9], but employ complex prefetch hardware and large tables to capture these patterns (assuming they remain stable). Those cases that conventional prefetching techniques do not cover well are handled in a straightforward manner with speculative precomputation because the prefetching threads are allowed to run decoupled from the main thread, and because addresses are calculated using code extracted from the original thread.

6. Speculative Precomputation using Basic Triggers

This section examines the performance gains of SP with basic triggers under two scenarios: in the first, ideal scenario, aggressive hardware support is assumed to gauge the upper bound for potential performance gains. In the second scenario, realistic implementation constraints for the Itanium family processors are taken into account, in which we leverage existing Itanium architectural features to facilitate implementation of SP without assuming aggressive hardware support.

6.1. Bounding Basic Trigger Performance

This subsection models two ideal SP configurations. Both only spawn speculative threads on the correct control path, but one does so when a trigger instruction reaches the rename stage (even though instructions which reach this stage are not guaranteed to be on the correct control path, we do not model wrong path spawning in this work), while the other waits until the commit stage (where the instruction is guaranteed to be on the correct path). In both cases, we assume aggressive and ideal hardware support for directly copying live-in values from the main thread's context to its child thread's context, i.e., one-cycle flash-copy. This allows the speculative thread to begin precomputation of a p-slice just one cycle after it is spawned.

Figure 6 shows the performance gains achieved through SP as the total number of hardware thread contexts is varied. For each benchmark, results are grouped into three pairs, corresponding to 2, 4 or 8 total hardware thread contexts. Within each pair, the configuration on the left models spawning speculative threads in the rename stage, and the one on the right models thread spawning in the commit stage.

Most benchmarks show gains from SP. For the most aggressive configuration (8 threads contexts, spawn in rename) the average speedup is 13.5%, and mst enjoys speedup of 32%. Gzip and health are the noticeable exceptions. As shown in Figure 3, having perfect delinquent loads only yields a 4% speedup for gzip (due to its high L2 hit rate). Thus, overhead from executing speculative threads hinders any performance gain. Health has potential for high speedup, but fails to achieve this speedup because it is not possible to sufficiently distance basic triggers from the targeted delinquent loads in its tight, pointer chasing loops. In Section 7, a new technique is introduced that overcomes this problem.

Increasing the number of hardware thread contexts results in opportunities for more speculation to be performed at runtime, reducing cancellation of thread spawning due to unavailable thread contexts. It is interesting to note that for all benchmarks which show benefits from SP, increasing thread contexts brings about more speedup. However, the contrary

is also true for benchmarks that suffer from SP; as the number of thread contexts is increased, gzip shows larger performance degradation.

6.2. Software-Based Speculative Precomputation

The previous subsection assumes a machine that employs ideal, one-cycle flash-copy between register files of the two thread contexts, permitting the non-speculative thread to spawn speculative threads instantly and without incurring any overhead cycles. Such an ideal machine may be difficult to implement for the Itanium family processors due to the cost of implementing a flash copy mechanism for such large register files. This reality motivates us to explore a less aggressive but more practical software-based SP (SSP) approach, which circumvents such hardware costs by directly taking advantage of the existing Itanium architectural features.

Before introducing the details of SSP, it is important to note that SP requires two basic mechanisms to support thread spawning regardless of implementation–a mechanism to bind a spawned thread to a free hardware context, and a mechanism to transfer necessary live-in values to the child thread. Both of these can be implemented using existing features of the Itanium processor family: on-chip memory buffers, which are used as spill area for the backing store of the Register Stack Engine (RSE) [8]; and the lightweight exception recovery mechanism, which is used to recover from incorrect control- and data speculations [14]. The result of using these existing Itanium features is a software-based approach for SP that does not require additional dedicated hardware. The details are described below.

Using LIB for Lightweight Live-in Transfer Without flash-copy hardware, one thread cannot directly access the registers of another thread, necessitating an intermediate buffer for transfer of live-in values from a parent thread to its child. Processors in the Itanium family contain special on-chip memory buffers for use as backing store for the RSE to host temporarily spilled registers. These buffers are architecturally visible, and can be accessed from every thread context. We allocate a portion of this buffer space and dedicate it as an intermediate buffer for passing live-in values from a parent thread to a child thread. We call this buffer space the Live-in Buffer (LIB).

The LIB is accessed through normal loads and stores, which are conceptually similar to spilling and refilling instructions except across register files of different thread contexts. The parent thread stores a sequence of values into the LIB before spawning the child thread, and the child thread, right after binding to a hardware context, loads the live-in values from the LIB into its context prior to executing the p-slice instructions. As shown in Table 3, there are fewer than 8 live-in values for most slices. Our processor features 4 loads/store units, permitting these live-in values to be passed from parent to child in only 4 cycles total.

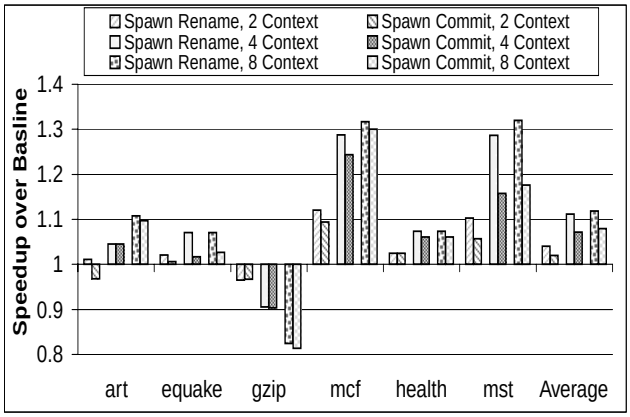

Figure 6. Speedup provided by basic triggers with ideal hardware assumptions.

Spawning Threads Via Lightweight Exception Recovery We can spawn a speculative thread and bind it to a free hardware context via the lightweight exception-recovery mechanism in the Itanium architecture. This mechanism uses the speculation check instructions to examine the results of user-level control- or data- speculative calculations to determine success or failure. Should failure occur, an exception surfaces and a branch is taken to a user defined recovery handler code within the thread, without requiring OS intervention. For example, when a chk.a (advanced load check) instruction detects that some store conflicts with an earlier advanced load, it will trigger branching into a recovery code [8] within the current program binary, and execute a sequence of instructions to repair the exception. Afterwards, the control branches back to the instruction following the one that raised the exception. We take advantage of this feature by introducing a new speculation check instruction, *chk.c* (available context check). The chk.c instruction raises an exception if a free hardware context is available for spawning a speculative thread. Otherwise, chk.c behaves like a nop.

A chk.c instruction is placed in the code wherever a basic trigger is needed. The recovery code simply stores the live-in state to the LIB, executes a spawn instruction to initiate the child thread and then returns. The child thread begins execution by loading the values from the LIB into its thread context.

There are two strengths to this approach over simply embedding the spawn code directly in the main program. First, the spawn code is only executed when a free thread context is actually available. Second, existing binaries can be easily modified to take advantage of this mechanism by changing a single instruction (for example, a nop) to a chk.c instruction and adding the recovery code at the end of the existing binary.

The SSP approach differs from the idealized hardware approach in two ways. First, spawning a thread is no longer instantaneous and will slow down the non-speculative thread by the time necessary to invoke and execute the exception handler. At the very minimum, invoking this exception han-

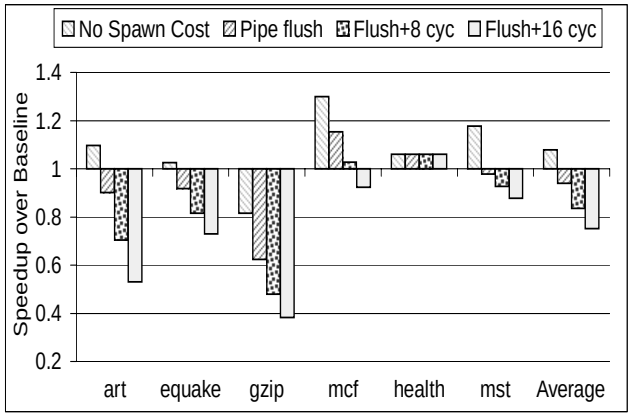

Figure 7. Speedup achieved by the software-based SP approach. Each bar corresponds to a different cost associated with spawning threads in the commit stage.

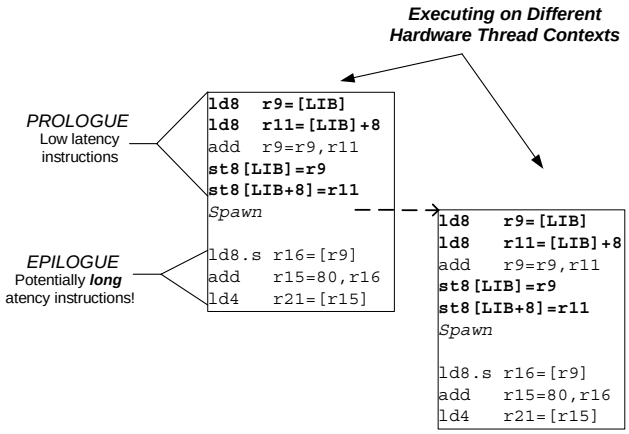

Figure 8. Runtime behavior of p-slice from Figure 5 after being enhanced to incorporate chaining triggers.

dler requires a pipeline flush. The second difference is that p-slices must be modified to first load their live-in values from the LIB, delaying the beginning of precomputation.

Performance of SSP Figure 7 shows the performance speedups achieved when using SSP for a processor with 8 hardware thread contexts. Four processor configurations are shown, each corresponding to differing thread spawning costs. The leftmost configuration is given for reference–speculative threads are spawned with no penalty for the non-speculative thread, but must still perform a sequence of load instructions to read their live-in values from the LIB. This configuration yields the highest possible performance under SSP because the main thread is still instantaneous in spawning a speculative thread. In the 3 remaining configurations, spawning a speculative thread causes the non-speculative thread's instructions following the chk.c to be flushed from the pipeline. In the configuration second from the left, this pipeline flush is the only penalty, while in the third and fourth configurations, an additional penalty of 8 and 16 cycles, respectively, is assumed for the cost of executing the recovery handler code.

These results fall far short of the ideal hardware results (see Figure 6), due primarily to the spawning overhead. The penalty of pipeline flush and the cost of performing the store instructions, both negatively affect the performance of the non-speculative thread. Two approaches to this problem are either choosing delinquent loads more judiciously by taking into account the overhead associated with spawning threads, or through incorporating additional hardware to accelerate thread spawning.

A third option is presented in the next section, which introduces a more aggressive form of SP that minimizes the overhead imposed on the non-speculative thread while still using the SSP approach. For the remainder of this paper, we assume SSP is still used to invoke basic triggers, and include its overhead in all our simulations (using the most conservative estimate of thread spawning overhead, a pipeline flush plus 16 cycles).

7. Speculative Precomputation with Chaining Triggers

Two problems limit performance gains from SP when only basic triggers are used. First, speculative threads are only spawned in response to progress made by the non-speculative thread. This means we are unable to spawn additional threads when the main thread stalls, when there is reduced fetch and execution contention. Second, to effectively prefetch data for delinquent loads, it is often necessary to precompute p-slices many loop iterations ahead of the non-speculative thread. Induction unrolling [13] was introduced for this purpose, but it increases the total number of speculative instructions executed without actually increasing the number of delinquent loads targeted. Executing more instructions also puts extra pressure on available hardware thread contexts because each speculative thread will occupy a thread context for a longer period.

7.1. Chaining Triggers

To overcome both problems described above we introduce a novel technique called chaining triggers, which allows one speculative thread to explicitly spawn another speculative thread. To illustrate the use of chaining triggers, we return to the sample loop from mcf shown in Figure 4. A key feature for applying chaining triggers to this loop (which was not effectively exploited with only basic triggers) is that the stride in the addresses consumed by load #1 is a dynamic invariant whose value is fixed for the duration of the loop.

Figure 8 shows how the basic p-slice from Figure 5 behaves at runtime after being enhanced to incorporate chaining triggers (notice the spawn instruction in the p-slice). Because the only loop-carried dependence affecting the delinquent loads is computed by the loop induction variable (which requires only a single add instruction), available parallelism can be aggressively exploited–immediately after computing the next address to be accessed by load #1, speculative threads are spawned to precompute for the next loop iteration. Thus, this use of chaining triggers makes it possible to precompute arbitrarily far ahead of the non-speculative thread, constrained only by the time necessary to compute necessary loop carried dependencies. In loops such as this one, where the loop-carried dependencies are computed early, chaining triggers can advance to future loop iterations much faster than the non-speculative thread. This feature makes it possible to achieve dramatically higher performance than with basic triggers alone.

Spawning a thread via a chaining trigger imposes significantly less overhead than a basic trigger because a chaining trigger requires no action from the main thread; instead the speculative thread directly stores values to the LIB and spawns child threads. For all benchmarks studied in this work except gzip, the vast majority of speculative threads are spawned from chaining triggers; for example, mcf contains a loop in which several hundred chaining triggers occur for each basic trigger.

7.2. Generation of Chaining Triggers

P-slices containing chaining triggers typically have three parts–a prologue, a spawn instruction for spawning another copy of this p-slice, and an epilogue. The prologue consists of instructions that compute values associated with a loop carried dependence, such as updates to a loop induction variable. The epilog consists of the instructions that actually produce the address accessed by the targeted delinquent load. The goal behind chaining trigger construction is for the prologue to be executed as quickly as possible, enabling additional speculative threads to be spawned as soon as possible.

A simple process can be used to add chaining triggers to basic p-slices that target delinquent loads within loops. The algorithm presented in Section 5.1 is augmented to track the distance between different instances of a delinquent load. If two instances of the same load consistently occur within some fixed sized window of instructions, a new p-slice is created which targets the load via chaining triggers. The prolog of the p-slice consists of instructions which modify values that are used in some future loop iteration to compute the address accessed by the delinquent load. The epilogue consists of the actual instructions within the loop used to compute the delinquent load address. Between the prologue and epilogue, a spawn instruction is inserted to spawn another copy of this p-slice.

7.3. Pending Slice Queue

It can be advantageous, especially in processors with few hardware thread contexts, to support a larger number of speculative threads than the number of total hardware thread contexts. This permits aggressive thread spawning, where 'overflow' speculative threads wait until a free thread context becomes available. We introduce a new hardware structure, called the Pending Slice Queue (PSQ). When a p-slice is spawned but all thread contexts are occupied, the p-slice is instead allocated an entry in the PSQ, if one is available. Thus, the sum of the total entries in the PSQ and the number of hardware contexts is the upper bound on the number of speculative threads that can exist at one time. Once allocated, a PSQ entry remains occupied until the thread is assigned to a hardware context. Waiting threads are allocated to hardware contexts using a FIFO policy.

The addition of the PSQ does not significantly increase processor complexity. The only necessary changes are to increase the size of the LIB and to add logic that chooses the next pending slice to assign to a thread context.

Because values stored in the LIB are no longer consumed immediately, it is necessary to increase its size to prevent useful values from being overwritten. However, the size of this buffer need not be excessively large–the number of register live-in values for all slices encountered in this research is never greater than 16. Assuming all of these values are 64 bits, a LIB statically partitioned for 16 threads would only require 2KBytes of storage.

It is in the use of the PSQ that the true benefit of the LIB becomes evident. Copying the live-in values from the parent thread to the LIB at spawn time ensures that the child thread will have valid live-in values to operate on when it eventually binds to a thread context, regardless of how long it is forced to wait in the PSQ.

7.4. Controlling Precomputation

Because the use of chaining triggers decouples thread spawning from progress made by the main thread, a control mechanism is necessary to prevent overly aggressive precomputation from getting too far ahead and evicting useful data from the cache before it has been accessed by the main thread. In addition, once the main thread leaves the scope of a p-slice, e.g. after exiting a pointer chasing loop or procedure, all speculative threads should be terminated to prevent useless prefetches.

We found that two simple mechanisms are sufficient to eliminate ineffective speculative threads. First, a thread that performs a memory access for which the hardware page table walker fails to find a valid translation, such as NULL pointer reference, is terminated; any chaining trigger executed afterwards in this p-slice is treated as a nop. This allows speculative threads to naturally "drain" out of the processor without spawning additional useless threads.

The second mechanism eliminates speculative threads

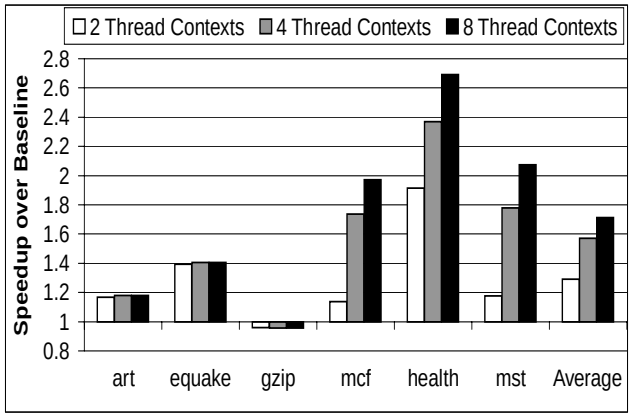

Figure 9. Speedup from Speculative Precomputation using both Basic and Chaining Triggers.

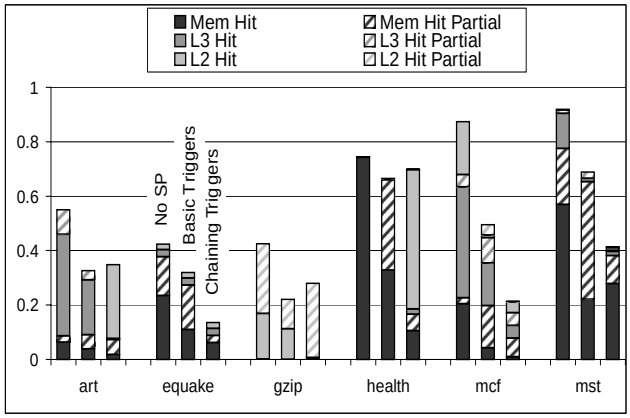

Figure 10. Where delinquent loads satisfied when missing in L1. Partial indicates access to a cache line already in transit to L1 from the indicated memory structure.

when the non-speculative thread leaves a section of the program. This can be achieved by adding an additional basic trigger that is equivalent to a speculative thread flush, which terminates all currently executing speculative threads and clears all entries in the PSQ. The thread flushing trigger can be inserted at the exit of scopes which spawn speculative threads.

Speculative precomputation provides maximal benefit when speculative threads are aggressive enough to fully cover memory latencies, but not so aggressive as to evict data out of the cache before they are accessed by the non-speculative thread. This delicate balance can be realized if speculative threads are permitted to advance far enough ahead of the non-speculative thread until their prefetches cover up the latency to main memory, but no further. Future work will address how to achieve this in a dynamic manner. In this research, we introduce a hardware structure to limit speculative threads to running only a fixed number (p-slice specific) of loop iterations ahead of the non-speculative thread.

The hardware structure is called the Outstanding Slice Counter (OSC). This structure tracks, for a subset of delinquent loads, the number of instances of delinquent loads for which a speculative thread has been spawned but for which the main thread has not yet committed the corresponding load. Each entry in the OSC contains a counter, the IP (instruction pointer) of a delinquent load and the address of the first instruction in a p-slice, which uniquely identifies the p-slice. This counter is incremented when the non-speculative thread retires the corresponding delinquent load, and is decremented when the corresponding p-slice is spawned. When a speculative thread is spawned for which the entry in the OSC is negative, the resulting speculative thread is instead allocated an entry in the PSQ, where it waits without being considered for assignment to a thread context until its counter becomes positive. Entries in the OSC are manually allocated in the exception recovery code associated with some basic trigger, and this research assumes a four entry, fully associative OSC.

7.5. Performance of SP with Chaining Triggers

Figure 9 shows the speedup achieved from SP using chaining triggers as the number of thread contexts is varied. Chaining triggers are highly effective at making use of available thread contexts when sufficient memory parallelism exists, resulting in average performance gains of 59% with 4 threads and 76% with 8 threads. In these results, the size of the PSQ is varied to ensure the number of entries in the PSQ plus the number of hardware thread contexts equals 16.

Though health did not benefit significantly from basic triggers (as shown in Figure 7), when using chaining triggers, the speedup is boosted to 169%, though opportunity for significant further improvement still exists (as shown in Figure 3). As noted in Section 6.1, health is dominated by tight pointer chasing loops, which are completely memory latency bound and exhibit no parallelism between loop iterations. By making use of chaining triggers, it is possible to precompute for multiple loop instances in parallel.

Figure 10 shows the breakdown of which level of the memory hierarchy is accessed by delinquent loads under three processor configurations — the baseline processor which does not use SP, a processor with 8 thread contexts which uses basic triggers and spawns them in the rename stage, and a processor with 8 thread contexts which uses both basic and chaining triggers. Also shown is the percentage of accesses to cache lines which were already in transit to L1 cache due to access by a prior load from the main thread or from a prefetch.

Table 4 shows prefetch statistics assuming the two SP configurations from above. The following information is shown: 1) Accuracy–percentage of prefetched lines accessed by a delinquent load before being evicted from L1 cache, 2) Partial–percentage of prefetched lines which are accessed before arriving at the L1 cache (partial loads are considered

Benchmark	Accuracy	Partial	Coverage	Spec Instr
art (B)	85.7%	14.2%	50.0%	61.7M
art (B+C)	35.0%	60.3%	94.8%	33.1M
equake (B)	94.0%	58.8%	59.4%	12.5M
equake (B+C)	91.4%	0%	62.0%	11.9M
gzip (B)	67.4%	9.3%	58.6%	20.2M
gzip (B+C)	79.8%	11.6%	96.7%	9.9M
health (B)	10.9%	86.7%	95.6%	20.7M
health (B+C)	24.0%	17.1%	22.6%	46.4M
mcf (B)	98.0%	37.8%	76.2%	22.5M
mcf (B+C)	92.4%	13.6%	88.2%	18.2M
mst (B)	90.7%	46.1%	66.6%	107.3M
mst (B+C)	74.5%	0%	56.8%	50.4M

Table 4. Statistics on prefetch accuracy and coverage of delinquent loads when assuming only basic triggers (B) and both basic and chaining triggers (B+C). Accuracy and Partial are given as percentages of total prefetches, Coverage is given as the percentage of total delinquent loads.

accurate), 3) Coverage–percentage of delinquent loads covered by SP, 4) Spec Instr–total number of speculative instructions executed.

In general, basic triggers provide high accuracy (for mcf, prefetch accuracy is 98%), but cover fewer loads than chaining triggers, and fail to significantly impact the number of loads which require access to main memory. For example, mcf only saw a 3% reduction in the number of loads that were satisfied from memory, although a large number of loads had their latency partially covered. Thus, basic triggers can be effective in targeting delinquent loads with relatively low latency, such as L2 hits, but are not likely to significantly help accesses to main memory.

Chaining triggers, on the other hand, achieve higher coverage and prefetch data in a much more timely manner, even data that requires access to main memory; when targeting mcf, the number of accesses to main memory by the non-speculative thread was reduced by more than 13% over the baseline. One somewhat anomalous case to this rule is health. Using chaining triggers, health appears to have extremely low prefetch accuracy. However, this is due primarily to the prefetching aggressiveness necessary to cover up the memory latency of its delinquent loads, causing useful prefetches to be evicted from L1 cache before being used. The high L2 cache hit rate attests to this. Thus, we see that one major advantage of chaining triggers over basic triggers is their ability to effectively target delinquent loads that are significantly far ahead of the non-speculative thread.

Chaining triggers also allow the processor to better utilize available thread contexts. Because they spawn additional speculative threads as soon as the p-slice prologue has been executed, chaining triggers are able to quickly populate all available hardware thread contexts. The simulation data attest to this–the average speedup from increasing the number of thread contexts from 4 to 8, when using chaining triggers,

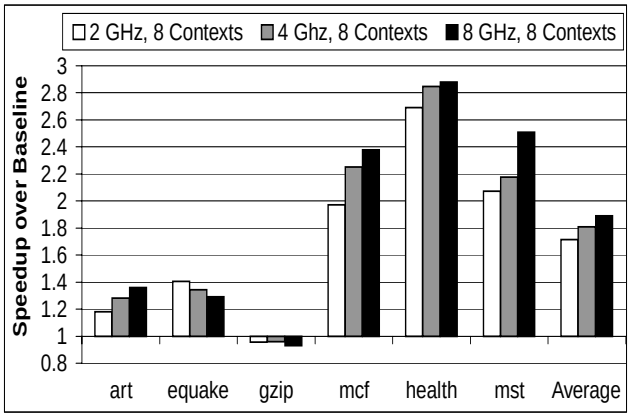

Figure 11. Speedup Provided by Basic and Chaining Triggers at 2, 4 and 8 GHz.

is 17%; when using basic triggers alone and spawning in the rename stage, this speedup is only 2%. In addition, this creates the opportunity for more outstanding prefetches to be in flight, effectively utilizing the bandwidth provided by non-blocking caches.

Mcf and mst both achieve large speedups over the baseline when the processor has four or eight total thread contexts. Reducing the total number of contexts to only two results in the smaller, but still significant, speedups of 13% for mcf and 12% for mst. Because p-slices from these two benchmarks contain multiple dependent loads, threads executing these p-slices are forced to stall when one of the loads misses in cache. With a larger number of thread contexts, other speculative threads can be scheduled 'around' the stalled thread onto the other available contexts. However, when only one thread context is available for speculative threads, a stall in a speculative thread prevents any further speculative threads from executing. We do not currently assume any preemption scheme for speculative threads, so the stalled thread will not relinquish its thread context to another speculative thread.

7.6. Effects of Pipeline Depth and Memory Latency

Figure 11 shows the performance gains provided by chaining triggers as clock frequency is increased, which implies higher memory latencies and longer pipelines (see Table 1). Results are shown for each benchmark for 3 clock frequencies (2GHz, 4GHz and 8GHz), with 8 total thread contexts. All speedups are shown relative to a respective baseline processor with the same clock frequency. Performance results for 2 and 4 thread contexts show similar trends.

Most benchmarks show continued speedups as clock frequency is increased. Equake is the primary exception to this. As shown in Figure 2, equake has a large number of delinquent loads and the 10 worst delinquent loads account for only about 60% of cache misses. As latency to memory increases, the performance impact of the 40% of misses not targeted by SP scales disproportionally, indicating that

at higher clock frequencies the criteria to select delinquent loads should be liberalized. However, the general trend is of increased effectiveness as the processor-memory gap widens.

8. Conclusion

This paper presents Speculative Precomputation (SP), a technique that allows a multithreaded processor to use spare hardware contexts to spawn speculative threads to prefetch data well in advance of the main thread. When the burden of spawning threads falls on the main non-speculative thread (via basic triggers), the potential speedup is as high as 30% assuming fast register copies between thread contexts. However, under more realistic assumptions, the potential speedup is significantly reduced. On the other hand, when the speculative threads can also spawn other speculative threads (via chaining triggers), dramatic speedups are possible on applications that have historically been resistant to prefetching techniques. These speedups are as high as 169% and average 76% over all benchmarks. This is achieved via a novel software based mechanism that can utilize existing Itanium processor features with very little additional hardware support.

9. Acknowledgements

We would like to thank Ralph Kling, Rakesh Ghiya, Perry Wang, Ryan Rakvic, Bohuslav Rychlik, Mauricio Serrano, Shih-wei Liao, Mary Xie, Sarita Adve, Justin Rattner, Bryan Black and Jim Dundas for their help in enhancing and validating our performance simulation, providing customized Itanium binaries and reviewing early versions of this paper. Additionally, we would like to thank the many referees of the previous versions of this paper for their extremely useful suggestions. This work was supported in part by a Focht-Powell fellowship, and NSF grant CCR-9808697.

References

[1] S. G. Abraham and B. R. Rau. Predicting load latencies using cache profiling. In *Hewlett Packard Lab, Technical Report HPL-94-110*, Dec. 1994.

[2] J. Bharadwajh. et al. The Intel IA-64 compiler code generator. In *IEEE Micro*, pages 44–53, Sept. 2000.

[3] M. Carlisle. Olden: Parallelizing programs with dynamic data structures on distributed-memory machines. In *PhD Thesis, Princeton University Department of Computer Science*, June 1996.

[4] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous subordinate microthreading (SSMT). In *26th Annual International Symposium on Computer Architecture*, pages 186–195, Oct. 1999.

[5] J. Emer. Simultaneous multithreading: Multiplying Alpha's performance. In *Microprocessor Forum*, Oct. 1999.

[6] M. D. Hill. Aspects of cache memory an instruction buffer performance. In *PhD Thesis, University of California, Berkeley*, 1987.

[7] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir. Introducing the IA-64 architecture. In *IEEE Micro*, pages 12–23, Sept. 2000.

[8] Intel Corporation. Intel IA-64 architecture software developer's manual.

[9] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[10] Y. Kim, M. Hill, and D. Wood. Implementing stack simulation for highly-associative memories (extended abstract). In *ACM Sigmetrics*, pages 212–213, May 1991.

[11] R. Krishnaiyer. et al. An advanced optimizer for the IA-64 architecture. In *IEEE Micro*, pages 60–68, Nov. 2000.

[12] A. Roth, A. Moshovos, and G. Sohi. Dependence based prefetching for linked data structures. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[13] A. Roth and G. Sohi. Speculative data-driven multithreading. In *Seventh International Symposium on High Performance Computer Architecture*, pages 37–48, Jan. 2001.

[14] H. Sharangpani and K. Aurora. Itanium processor microarchitecture. In *IEEE Micro*, pages 24–43, Sept. 2000.

[15] Y. Song and M. Dubois. Assisted execution. In *Technical Report CENG 98-25, Department of EE-Systems, University of Southern California*, Oct. 1998.

[16] SPEC. SPEC cpu2000 documentation. In *http://www.spec.org/osg/cpu2000/docs/*.

[17] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–268, Nov. 2000.

[18] D. Tullsen. Simulation and modeling of a simultaneous multithreaded processor. In *22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[19] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[20] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[21] R. Uhlig, R. Fishtein, O. Gershon, I. Hirsh, and H. Wang. SoftSDV: A presilicon software development environment for the IA-64 architecture. In *Intel Technology Journal*, 4th Quarter 1999.

[22] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. In *25th Annual International Symposium on Computer Architecture*, pages 238–249, June 1998.

[23] H. Wang et al. A conjugate flow processor. In *Docket No. 884.225US1. Patent Pending*, May 2000.

[24] C. Young, N. Gloy, and M. D. Smith. A comparative analysis of schemes for correlated branch prediction. In *22nd Annual International Symposium on Computer Architecture*, pages 276–286, May 1995.

[25] C. Zilles and G. Sohi. Understanding the backward slices of performance degrading instructions. In *27th Annual International Symposium on Computer Architecture*, pages 172–181, June 2000.